Introdução à Computação II – AULA 14

BCC Noturno - EMA896115B

Prof. Rafael Oliveira
olivrap@gmail.com

Universidade Estadual Paulista
“Júlio de Mesquita Filho”
UNESP

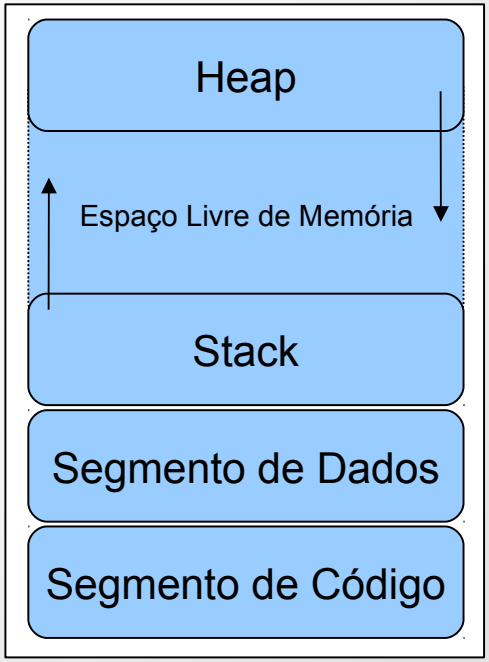
Rio Claro 2014 (Sem 2)

Objetivos

- Entender o que são e como usar:
  - Gerenciamento de Memória
  - Alocação Dinâmica em C

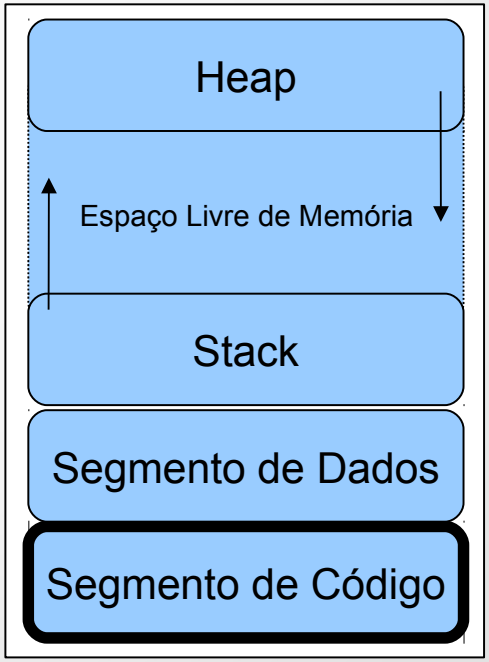
Gerenciamento de Memória

- A memória utilizada por um programa de computador é dividida em:
  - Segmento de Código
  - Segmento de Dados
  - Stack
  - Heap
- Cada programa em execução tem seu próprio Seg. Código, Seg. De Dados, Stack e Heap



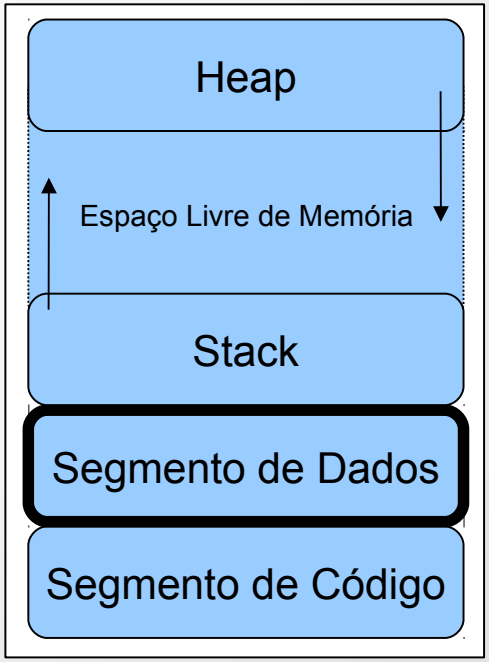
Gerenciamento de Memória

- O **Segmento de Código** é a parte da memória que armazena o “código de máquina” do programa
- É estático em tamanho e conteúdo (de acordo com o executável)
- Geralmente, o bloco de segmento de código é somente leitura
  - As instruções do programa compilado e em execução não pode ser alterado



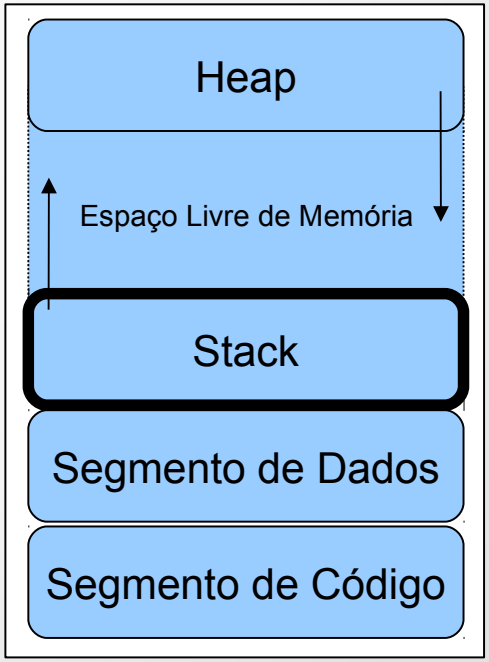
Gerenciamento de Memória

- O **Segmento de Dados** é a parte da memória que armazena as “variáveis globais” e “variáveis estáticas” inicializadas no código do programa
- O tamanho do segmento é calculado de acordo com os valores das variáveis definidas
- O acesso é de leitura-escrita
  - Os valores das variáveis neste segmento podem ser alterados durante a execução do programa



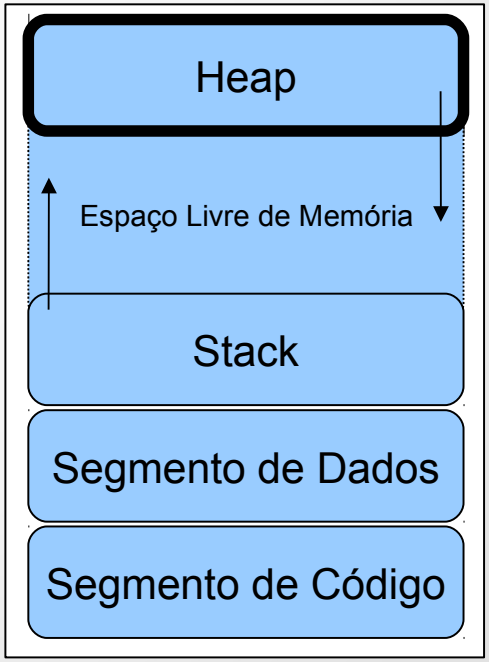
Gerenciamento de Memória

- O **Stack** é a parte da memória que armazena as "variáveis locais" e chamadas de funções do programa
- Usa a estratégia LIFO (last-in-first-out) para gerenciar a entrada/saída de dados na memória
- O tamanho da Stack é variável e depende do sistema operacional e compilador utilizados
- Utilizar mais memória Stack do que disponível provoca um erro de execução: "*stack buffer overflow*"



Gerenciamento de Memória

- **Heap** é um espaço reservado para *alocação dinâmica* de memória dos programas
- Memória alocada dinamicamente pode ser usada e liberada a qualquer momento
- A linguagem C fornece funções próprias para lidar com alocação dinâmica de memória
  - malloc
  - calloc
  - free



Malloc (Memory Alocation)

- Função para requisitar alocação dinâmica de memória
- Recebe como parâmetro o tamanho em bytes de memória a ser alocada
- Retorna um ponteiro para o início da memória alocada
  - Se ocorrer algum erro de alocação de memória, a função malloc retorna NULL

Malloc (Memory Alocation)

- Exemplo:
  - Alocar dinâmicamente um espaço de memória para 30 inteiros (vetor).

```
int *vet = NULL;
vet = (int*) malloc(30 * sizeof(int));
```

Tipo de dados que o ponteiro irá receber

Número de inteiros que desejamos alocar

Tamanho em bytes do tipo "int" na linguagem C

- É possível alocar memória para qualquer tipo de dados, incluindo as estruturas (structs)

Calloc (Cleared Alocation)

- Faz alocação de memória em blocos e inicializa a memória alocada com zero (cleared)
  - A memória alocada com malloc não é inicializada (contém lixo). Fica a cargo do programador inicializar a memória alocada.
- Recebe como parâmetro o número de elementos a serem alocados e o tamanho em bytes dos elementos
- Retorna um ponteiro para o início da memória alocada
  - Se ocorrer algum erro de alocação de memória, a função malloc retorna NULL

Calloc (Cleared Alocation)

- Exemplo:
  - Alocar dinâmicamente um espaço de memória para 30 inteiros (vetor).

```
int *vet = NULL;
vet = (int*) calloc(30, sizeof(int));
```

Tipo de dados que o ponteiro irá receber

Número de inteiros que desejamos alocar

Tamanho em bytes do tipo “int” na linguagem C

Free

- Toda memória alocada deve ser LIBERADA.
  - É uma boa prática de programação
- A função free recebe como entrada o ponteiro para a memória alocada
- Exemplo:

```
/* alocando memória */
int *vet = NULL;
vet = (int*) malloc(30 * sizeof(int));

/* liberando memória */
free(vet);
```

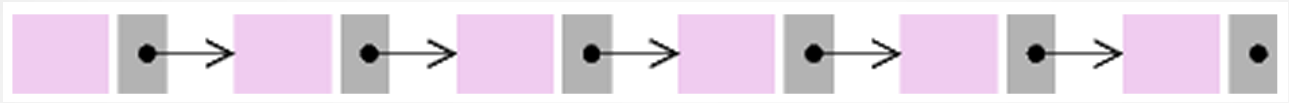
Alocando memória com sucesso

- Em algumas situações, pode não haver mais memória disponível para o programa
  - É recomendado que o programador verifique sempre se a memória foi alocada com sucesso!
- Uma forma usual é escrever uma função para auxiliar a alocação dinâmica de memória

```
int *alocar_memoria(int tamanho) {
  int *new_mem;
  new_mem = malloc(tamanho*sizeof(int));
  if (new_mem == NULL){
    printf("Nao foi possivel alocar memória");
    exit(1);
  }
  return new_mem;
}
```

Lista Encadeada

- Estruturas de dados lineares



- Flexibilidade em comparação aos vetores
- O programador define as regras para:
  - Inserção, Remoção, Busca e Atualização
- Algumas listas especializadas
  - Fila: primeiro a entrar, primeiro a sair
  - Pilha: último a entrar, primeiro a sair

Lista Encadeada

- Uma lista encadeada simples
  - Cada elemento possui um valor
  - Um elemento “aponta” para próximo elemento da lista
  - O último elemento aponta para NULL
- Estrutura:

```
struct No{
    int valor;
    struct No *p_prox;
};
```

Lista Encadeada

- Iniciando uma lista

```
void iniciar_lista(struct No **p_Raiz){
   *p_Raiz = NULL;
}
```

  - O primeiro elemento da lista é denominado Raiz
  - A lista sempre começa vazia
    - Raiz aponta para NULL

Lista Encadeada

- Inserção de elementos
  - Novos elementos podem ser inseridos qualquer posição da lista
  - Alocação dinâmica para cada novo elemento
  - Exemplo: inserção no início da lista (desloca a raiz para frente)

```
void inserir (struct No **p_Raiz, int valor){
    struct No *p_Novo;
    p_Novo = (struct No *) malloc(sizeof(struct No)));
    p_Novo->valor = valor;
    if(*p_Raiz == NULL){
        *p_Raiz = p_Novo;
        p_Novo->p_prox = NULL;
    }else{
        p_Novo->p_prox = *p_Raiz;
        *p_Raiz = p_Novo;
    }
}
```

Lista Encadeada

- Percorrendo a lista
  - Utiliza os ponteiros para “caminhar” entre os elementos da lista
  - Exemplo:
    - Percorrendo da raiz até o último elemento da lista

```
void percorrer (struct No *p_Raiz){
    if(p_Raiz == NULL) printf("\nLista vazia");
    else{
        struct No *p_atual;
        p_atual = *p_Raiz;
        while(p_atual != NULL){
            printf("%d", p_atual->valor);
            p_atual = p_atual->p_prox;
        }
    }
}
```

Lista Encadeada

- Removendo elementos
  - Elementos removidos devem ser liberados na memória (free)
  - Nenhum elemento da lista deve apontar para um elemento removido
  - Exemplo: removendo o início da lista

```
void remover (struct No **p_Raiz){
    if(*p_Raiz == NULL) printf("\nLista ja esta vazia\n");
else{
        struct No *p_atual;
        p_atual = *p_Raiz;
        *p_Raiz = (*p_Raiz)->p_prox;
        free(p_atual);
    }
}
```

Exercício

- Considere a estrutura abaixo e as operações de lista encadeada desta aula

```
struct No{
    int valor;
    struct No *p_prox;
};
```

- Implemente uma função para buscar um elemento (por meio do campo “valor) e removê-lo de qualquer posição da lista